# Dissecting Tendermint

Yackolley Amoussou-Guenou, Antonella del Pozzo, Maria Potop-Butucaru, Sara Tucci-Piergiovanni

# Dissecting Tendermint

Yackolley Amoussou-Guenou[1,2], Antonella Del Pozzo[1], Maria Potop-Butucaru[2], and Sara Tucci-Piergiovanni[1]

[1] CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
[2] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

**Abstract.** In this paper we analyze Tendermint, proposed in [12], one of the most popular blockchains based on PBFT Consensus. Our methodology consists in identifying the algorithmic principles of Tendermint necessary for a specific system model. The current paper dissects Tendermint under two communication models: synchronous and eventually synchronous ones. This methodology allowed to identify bugs in preliminary versions of the protocol and to prove its correctness under the most adversarial conditions: an eventually synchronous communication model under Byzantine faults. The message complexity of Tendermint is $O(n^3)$.

**Keywords:** BFT Consensus · Blockchain · Tendermint · Complexity

## 1 Introduction

A blockchain is a distributed ledger implementing an append-only list of blocks chained to each other, it serves as an immutable and non repudiable ledger in a system composed of untrusted processes. The append operation needs to preserve the chain shape of the data structure, leading to the necessity to have a mechanism allowing processes to agree on the next block to append. Bitcoin blockchain, for example, employs the proof-of-work mechanism [19], that is, processes willing to append a new block have to solve a crypto-puzzle and the winning process will append the new block. While this mechanism does not require a real coordination between the processes participating to the Bitcoin system, it might lead to inconsistencies. Indeed, if more than one process solves the crypto-puzzle to extend the same last block then processes may have blockchains with different suffix as long as the conflict is unsolved.

In blockchain systems area the recent tendency is to privilege solutions based on distributed agreement than proof-of-work. This is motivated by the fact that the majority of proof-of-work based solutions such as Bitcoin or Ethereum are energetically not viable when efficiency is targeted. Moreover proof of work solutions guarantee the existence of an unique chain only with high probability which is the major drawback for using blockchains in industrial applications. That is, forks even though they are rare do still happen with an impact on the consistency guarantees offered by the system and consensus algorithms play an important role to prevent inconsistencies. In [8] the authors proved that consensus [27] is necessary in order to avoid forks. Therefore, alternatives to proof-of-work have

been recently considered and interestingly, the research in blockchain systems revived a branch of distributed systems research: Byzantine fault-tolerant protocols having PBFT consensus protocol as ambassador. It should be noted that PBFT solutions cannot be used in permissionless settings if the number of participants to the agreement is not known in advance. That is, in permissionless settings, for each block, a subset of processes (called validators in Tendermint) runs a Byzantine fault-tolerant consensus algorithm to propose the next block to be appended to the blockchain. All the existing solutions for PBFT consensus use the number of validators as hardcore information in their algorithm.

*Related Work.* In the blockchain realm, there exist several Byzantine Fault Tolerant Consensus based blockchain proposals (e.g., [3, 9, 16, 17], and [23]).

The consensus problem, as proved in the seminal FLP paper [21], cannot be solved in an asynchronous message-passing system (when there are no upper bounds on the message delivery delay) in the presence of one faulty (crash) process. Moreover, in [27], the authors prove that consensus cannot be solved in presence of $f$ Byzantine faulty processes if the overall number of processes $n$ is less than $3f + 1$ in a synchronous message-passing system (where the message delivery delay is upper bounded). In between those impossibility results, it is still possible to solve consensus in an asynchronous setting, either adding randomness [11] (which also proved the impossibility result for $n \leq 3f$ for any asynchronous solution) or partial synchrony as in Dwork et al. [18] (DLS) where BFT Consensus is solved an eventual synchronous message-passing system (there is a time $\tau$ after which there is an upper bound on the message delivery delay). DLS preserves safety during the asynchronous period and the termination only after $\tau$, when the message transfer delay becomes bounded. The message complexity of this protocol is $O(n^4)$ per epoch and it needs $O(n)$ epochs before deciding. Finally, Castro and Liskov proposed PBFT [14], a leader-based protocol that optimizes the performances of the previous solution. If the leader is correct the complexity boils down to $O(n^2)$. Otherwise, a view change mechanism takes place, to change the leader and resume the computation. The view-change is used to avoid that, in case of faulty leader, if some correct process decides on a value $v$, the other correct processes cannot decide on a value $v' \neq v$ when the new leader proposes a new value. Such mechanism implies that when a leader is suspected to be faulty, all processes have to collect enough evidences for the view-change. That is, the view-change message contains at least $2f + 1$ signed messages and these messages are sent from at least $2f + 1$ processes which yields a message complexity of $O(n^2)$. These messages are then sent to all processes, the view-change has then $O(n^3)$ message complexity. Since the protocol terminates when there is a correct leader, which may happen for the first time in epoch $f + 1$, then in the worst case scenario it has a message complexity of $O(n^4)$. Interestingly, Tendermint as well as similar recent approaches e.g [2] use an alternative mechanism for leader replacement that allows to drop message complexity to $O(n^3)$. Basically, processes instead of exchanging all the messages they already delivered (used previously to trigger a view change), locally keep track of potentially decided values.

*Our Contribution.* In this paper we analyze Tendermint proposed in [12] as one of the most promising but not fully analyzed blockchain protocols that implements Byzantine fault tolerant consensus. Tendermint targets an eventual synchronous system [18], which means that safety has to be guaranteed in the asynchronous periods and liveness in synchronous ones, when a subset of processes can be affected by Byzantine failures. To analyze the protocol, we dissect Tendermint identifying the techniques used to address different challenges in the considered system model: synchronous round-based communication model and eventual synchronous communication model. For each type of model we provide the corresponding algorithm (a variant of Tendermint [12]) and compute its complexity. Interestingly, and contrary to the classical view-changed based approaches, message complexity in the worst case scenario is $O(n^3)$. This is because processes, instead of exchanging all the messages they already delivered, locally keep track of potentially decided values to preserve the safety, hence reducing the message complexity. In the same spirit, HotStuff [2] (a concurrent proposal) incurs the same message complexity, sharing with Tendermint a linear proposer replacement. Note as well that the proposed methodology allowed us to identify bugs (see [5]) in the preliminary versions of the protocol ([12, 26]).

This paper and [6] target two different consensus algorithms that are core of two different releases of Tendermint blockchain. In [6] the authors reverse-engineered and then formalized the Tendermint blockchain protocol implemented initially by the Tendermint Foundation [31]. [6] allowed to identify several bugs in the initial version of Tendermint implementation (see [5]). Moreover, we proved that the termination property cannot be guaranteed in general, and hence an additional assumption on the execution is needed to solve Consensus. After the publication of our findings, Tendermint foundation proposed a new algorithm, [12], that is currently implemented as consensus-core for the new release of Tendermint. The new version of the protocol claimed to include new mechanisms that removed the need of additional assumptions in order to guarantee the termination. The pseudo-code proposed in [12] and further implemented by Tendermint foundation still had some bugs at the time when we started to analyse it, which we reported [30].

In order to help practitioners, and in particular Tendermint foundation, to detect easily their errors and compare with the existing state of the art, in this paper we decided to have a bottom up approach by identifying the minimal building blocks a PBFT-like protocol should include in order to solve consensus function on the considered system and communication model (going from synchronous to eventually synchronous) and the behavior of Byzantine nodes. We used Tendermint as case study and identified the mechanisms needed by the protocol in order to be correct. Our study resulted in three variants of the protocol for which we analyzed the correctness and the complexity. In this paper, we included two of the three algorithms (we decided to left aside the trivial one where Byzantines have a symmetrical behavior and the communication is synchronous). Moreover, the complexity analysis proposed in our paper may help

both practitioners and academics to compare Tendermint to the state of the art which was an open question so far.

## 2 Model

The system is composed of an infinite set $\Pi$ of sequential processes, namely $\Pi = \{p_1, \dots\}$; *Sequential* means that a process executes one step at a time. This does not prevent it from executing several threads with an appropriate multiplexing. As local processing time are negligible with respect to message transfer delays, they are considered as equal to zero.

**Arrival model.** We assume a *finite arrival model* [4], i.e. the system has infinitely many processes $\Pi$ but each run has only finitely many. The size of the set $\Pi_\rho \subset \Pi$ of processes that participate in each system run is not a priori-known. We also consider a finite subset $V \subseteq \Pi_\rho$ of validators. The set $V$ may change during any system run and its size $n$ is a-priori known. A process is promoted in $V$ based on a so-called merit parameter, which can model for instance its stake in proof-of-stake blockchains. Note that in the current Tendermint implementation, it is a separate module included in the Cosmos project [25] that is in charge of implementing the selection of $V$.

**Failure model.** There is no bound on processes that can exhibit a Byzantine behaviour [29] in the system, but up to $f$ validators can exhibit a Byzantine behaviour at each point of the execution. A Byzantine process is a process that behaves arbitrarily. A process (or validator) that exhibits a Byzantine behaviour is called *faulty*. Otherwise, it is *non-faulty* or *correct* or *honest*. To be able to solve the consensus problem, we assume that $f < n/3$ and more precisely we consider $n = 3f + 1$.

**Communication model.** Processes communicate by exchanging messages through an eventually synchronous network [18]. *Eventually Synchronous* means that after a finite unknown time $\tau > 0$ there is a bound $\delta$ on the message transfer delay. When $\tau = 0$ the network is *synchronous*.

In the following we assume the presence of a *broadcast primitive*. A process $p_i$ by invoking the primitive broadcast$(\langle TAG, m\rangle)$ broadcasts a message, where $TAG$ is the type of the message, and $m$ its content. To simplify the presentation, it is assumed that a process can send messages to itself. The primitive broadcast() is a best effort broadcast, which means that when a correct process broadcasts a value, eventually all the correct processes deliver it. A process $p_i$ receives a message by executing the primitive delivery(). Messages are created with a digital signature, and we assume that digital signatures cannot be forged. When a process $p_i$ delivers a message, it knows the process $p_j$ that created the message.

Let us note that the assumed broadcast primitive in an open dynamic network can be implemented through *gossiping*, i.e. each process sends the message to current neighbors in the underlying dynamic network graph. In these settings the finite arrival model is a necessary condition for the system to show eventual synchrony. Intuitively, a finite arrival implies that message losses due to topology

changes are bounded, so that the propagation delay of a message between two processes not directly connected can be bounded [10, 28].

**Round-based execution model.** We assume that each correct process evolves in rounds. A *round* consists of three phases, in order: (i) a *Send* phase, where the process broadcasts messages computed during the last round, or a default messages for the first round; (ii) a *Delivery* phase where the process collects messages sent during the current and previous rounds; and (iii) a *Compute* phase where the process uses the messages delivered to change its state. At the end of a round a process exits from the current round and starts the next round. Each round has a finite duration, we consider the Send and the Compute phase as being atomic, they are executed instantaneously, but not the Delivery phase. In a synchronous network, we assume the duration of the Delivery phase, and so of the round is $\delta$. We assume that processes have no access to a global clock but have access to local clocks, these clocks might not be synchronized with each other but are allowed to have bounded clock skew.

**Problem definition.** In this paper we analyze the correctness of Tendermint protocol with respect to the consensus specification: **Termination**, every correct process eventually decides some value; **Integrity**, no correct process decides twice; **Agreement**, if there is a correct process that decides a value $v$, then eventually all the correct processes decide $v$; **Validity[13, 15]**, a decided value is valid, it satisfies the predefined predicate denoted valid().

## 3 Tendermint Algorithms

Tendermint BFT Consensus protocol [12, 26, 31] is a variant of PBFT consensus, at the core layer of the Tendermint blockchain.

The algorithm follows the rotating coordinator paradigm i.e., for each new block to be appended there is a proposer, chosen among the validators, that proposes the block. If the block is not decided then a new proposer is selected and so on, until a block is decided by all the correct validators and consensus terminates. In the following we present variants of [12] in synchronous and eventual synchronous communication models.

*Basic principles of the protocol.* Each block in the blockchain is characterized by its height $h$, which is the distance in terms of blocks from the genesis block, which is at height 0. For each new height, the two protocols (Algorithm 2 for the synchronous case and Algorithm 4 for the eventual synchronous case) share a common algorithmic structure, they proceed in *epoch*s, and each epoch $e$ consists in three rounds: the *PRE-PROPOSE* round; the *PROPOSE* round; and the *VOTE* round. During the PRE-PROPOSE round, the proposer pre-proposes a value $v$ to all the other validators. During the PROPOSE round, if a validator accepts $v$ then it proposes such value. If a validator receives *enough* proposals for the same value $v$ then it votes for $v$ during the VOTE round. Finally, if a validator receives *enough* votes for $v$, it decides on $v$. In this case, *enough* means at least $2f + 1$ occurrences of the same value from $2f + 1$ different validators and

from each validator only the first value delivered for each round is considered, (cf. Algorithm 1).

If the proposer is correct then it pre-proposes the same value to all the $2f+1$ correct validators. All the $2f+1$ correct validators propose such value, it follows that all the $2f+1$ correct validators vote for such value and decide for it. If the proposer is Byzantine it can pre-propose different values to different correct validators, creating a partition in the proposal value set collected by validators. Depending on what the remaining Byzantine validators do, some correct validators may decide on a value $v$ and some other may not[3], then a new epoch starts. In order to not violate the agreement property, validators that have not decided yet in the previous epoch must only decide for $v$, for this reason validators, before vote for some value $v$, lock on that value, i.e., they will refuse to propose a further pre-proposed value different than $v$.

*Information from one epoch to the next.* $lockedValue$ and $validValue$ variables[4] carry the potentially decided value from one epoch to the next one. The $lockedValue$ idea is the following. If one correct validator decides on $v$, it means that it collected $2f+1$ votes for $v$ during the VOTE phase, since there are at most $f$ Byzantine validators thus there are at least $f+1$ correct validators that voted for $v$ and those validators must not vote for any other different value than $v$. For this reason if a validator delivers $2f+1$ proposals for $v$ during the PROPOSE round it sets its $lockedValue$ to $v$. Since each new pre-proposed value $v'$ is proposed if $v'$ is equal to $lockedValue$ or $validValue$ (not true for at lest $f+1$ correct validators that set $lockedValue$ to $v$), then there can be at most $2f$ possible proposals for $v'$ that are not enough to lock and vote for $v'$, i.e., it is not possible to decide for any value different than $v$. On the other side, if no correct validator decided yet, Byzantine faulty validators may force different correct validators to lock on different values. Let us consider a scenario where the proposer is Byzantine and proposes $v$ to $f+1$ correct validators and then $f$ Byzantine validators make $x \leq f$ of them lock on $v$ and a similar scenario can happen with another value $v'$ so that we can have different correct validators, let us say $y \leq f$ locked on a different value. If any new pre-proposal is checked only against the $lockedValue$ then a correct validator locked on a value $v$ refuses (does not propose) all values different from $v$, it means that when some correct validator is locked, the proposer needs to propose some of the value on which the correct validators are locked on, but such value, in order to be accepted cannot be checked only against the $lockedValue$ because we may never have enough correct validators proposing such value. For this reason validators keep track of the $validValue$ and by construction of the algorithm all correct validators have the same $validValue$ at the end of the epoch (in the synchronous period). Such value is then used to set the value to pre-propose and it is further used along with $lockedValue$ to accept or not a pre-proposed value.

[3] Since there are $3f+1$ validators, there cannot be two different values that collect $2f+1$ distinct votes in the same epoch.

[4] $validValue$ was not present in the previous version of Tendermint [26], that was suffering from the Live Lock bug [1].

**Algorithm 1** Messages management for validator $p_i$

```
1: upon ⟨TYPE, h, e, message⟩ from validator p_j do
2:     if ∄c : (⟨TYPE, h, e, c⟩, p_j) ∈ messagesSet then
3:         messagesSet_i ← messagesSet_i ∪ (⟨TYPE, h, e, message⟩, p_j)
```

**Messages syntax.** When the validator $p_i$ broadcasts a message $\langle TAG, h, e, m\rangle$, where $m$ contains a value $v$, we say that $p_i$ pre-proposes, proposes or votes $v$ if $TAG$=`PRE-PROPOSE`, $TAG$=`PROPOSE`, $TAG$=`VOTE`, respectively.

**Variables and data structures.** $h$ is an integer representing the consensus instance the validator is currently executing. $e_i$ is an integer representing the epoch where the validator $p_i$ is, we note that for each height, a validator may have multiple epochs. $decision_i$ is the decision of validator $p_i$ for the consensus instance $h$. $proposal_i$ is the value the validator $p_i$ proposes. $vote_i$ is the value the validator $p_i$ votes. $lockedValue_i$ stores a value which is potentially decided by some other validator. If validator $p_i$ delivers more than $2f + 1$ proposes for the same value $v$ during its PROPOSE round, it sets $lockedValue_i$ to $v$. $validValue_i$ stores a value which is potentially decided by some other validator. If the validator $p_i$ delivers at least $2f + 1$ proposes for the same value $v$ (from different validators) whether during its PROPOSE round or its VOTE round, it sets $validValue_i$ to $v$. $validValid_i$ is the last value that a validator delivered at least $2f + 1$ times, and can be different than $lockedValue_i$. The latter two variables are used as follows: if $p_i$ is the next proposer then $p_i$ pre-proposes $validValid_i$ if different from $nil$. Otherwise, if $p_i$ is a validator, it checks the new pre-proposal against $lockedValue_i$ and $validValid_i$ if those are different from $nil$.

**Functions.** We denote as $Value$ the set containing all blocks, as $MemPool$ the set containing all the transactions, and as $Messages$ the set containing all messages.

- proposer : $Height \times Epoch \to V \subseteq \Pi_\rho$ is a deterministic function which gives the proposer out of the validators set for a given epoch at a given height in a round robin fashion.
- valid : $Value \to Bool$ is an application dependent predicate that is satisfied if the given value is valid w.r.t. the blockchain. If there is a value $v$ such that $\mathsf{valid}(v) = \mathsf{true}$, we say that $v$ is valid. Note that we set $\mathsf{valid}(nil) = \mathsf{false}$.
- getValue() return a valid value.
- sendByProposer : $Height \times Epoch \times Value \to Bool$ is an predicate that gives true if the given value has been pre-proposed by the proposer of the given height during the given epoch.
- $2f + 1$ : $\mathcal{P}(\texttt{Messages}) \to Bool$: checks if there are at least $2f + 1$ proposals (resp. votes) in the given set of messages.

Everything defined above is common to the two algorithms. In each section we specify the data structures relative to a specific version of the algorithm.

**Algorithm 2** Simplified Algorithm part 1 for height $h$ executed at validator $p_i$

```
1:  Initialization:
2:    e_i := 0                                                    /* This current epoch number */
3:    decision_i := nil                          /* This variable stocks the decision of the validator p_i */
4:    lockedValue_i := nil; validValue_i := nil
5:    proposal_i := getValue() /* This variable stocks the value the validator will (pre-)propose */
6:    v_i := nil                                 /* Local variable stocking the pre-preposal if delivered */
7:    vote_i := nil

8:  Round PRE-PROPOSE(e_i):
9:    Send phase:
10:     if decision_i ≠ nil then
11:       ∀v, p_j : (⟨VOTE, h, e_i, v⟩, p_j) ∈ messagesSet_i, broadcast⟨VOTE, h, e_i, v⟩
12:       return
13:     if proposer(h, e_i) = p_i then
14:       broadcast ⟨PRE − PROPOSE, h, e_i, proposal_i⟩ to all validators
15:   Delivery phase:
16:     while (timerPrePropose not expired) do
17:       if ∃v : sendByProposer(h, e_i, v) then
18:         v_i ← v                                   /* v is the value sent by the proposer */
19:   Compute phase:
20:     if !valid(v_i) then
21:       proposal_i ← nil                            /* Note that valid(nil) is set to false */
22:     else
23:       if validValue_i = nil ∨ v_i ∈ {lockedValue_i, validValue_i} then
24:         proposal_i ← v_i
25:       else
26:         proposal_i ← nil
```

### 3.1 Byzantine Synchronous System

In Algorithms 1 - 3 we describe the algorithm to solve consensus in a synchronous system in presence of Byzantine failures. The algorithm proceeds in 3 rounds for any given epoch at height $h$:

- Round PRE-PROPOSE (lines 8 - 26, Algorithm 2): If the validator $p_i$ is the proposer of the epoch, it pre-proposes its proposal value, otherwise, it waits for the proposal from the proposer. The proposal value of the proposer is its $validValue_i$ if $validValue_i \neq nil$. If a validator $p_j$ delivers the pre-proposal from the proposer of the epoch, $p_j$ checks the validity of the pre-proposal and if to accept it with respect to the values in $validValue_i$ and $lockedValue_i$. If the pre-proposal is accepted and valid, $p_j$ sets its proposal $proposal_j$ to the pre-proposal, otherwise it sets it to $nil$.
- Round PROPOSE (lines 1 - 13, Algorithm 3): During the PROPOSE round, each validator broadcasts its proposal, and collects the proposals sent by the other validators. After the Delivery phase, validator $p_i$ has a set of proposals, and checks if $v$, pre-proposed by the proposer, was proposed by at least $2f + 1$ different validators, if it is the case, and the value is valid, then $p_i$ sets $vote_i, validValue_i$ and $lockedValue_i$ to $v$, otherwise it sets $vote_i$ to $nil$.
- Round VOTE (lines 14 - 32, Algorithm 3): In the round VOTE, a correct validator $p_i$ votes $vote_i$ and broadcasts all the proposals it delivered during the current epoch. Then $p_i$ collects all the messages that were broadcast. First $p_i$ checks if it has delivered at least $2f + 1$ of proposal for a value $v'$ pre-proposed by the proposer of the epoch, in that case, it sets $validValue_i$

**Algorithm 3** Simplified Algorithm part 2 for height $h$ executed at validator $p_i$

```
1: Round PROPOSE(e_i):
2:   Send phase:
3:     if proposal_i ≠ nil then
4:       broadcast ⟨PROPOSE, h, e_i, proposal_i⟩ to all validators
5:   Delivery phase:
6:     while (timerPropose not expires) do{}                    /* Collect messages */
7:   Compute phase:
8:     if ∃v : 2f + 1⟨PROPOSE, h, e_i, v⟩ ∧ valid(v) ∧ sendByProposer(h, e_i, v) then
9:       lockedValue_i ← v
10:      validValue_i ← v
11:      vote_i ← v
12:    else
13:      vote_i ← nil

14: Round VOTE(e_i):
15:   Send phase:
16:     ∀v, p_j : (⟨PROPOSE, h, e_i, v⟩, p_j) ∈ messagesSet_i, broadcast⟨PROPOSE, h, e_i, v⟩
17:     if vote_i ≠ nil then
18:       broadcast ⟨VOTE, h, e_i, vote_i⟩
19:   Delivery phase:
20:     while (timerVote not expires) do{}                       /* Collect messages */
21:   Compute phase:
22:     if ∃v' : 2f + 1⟨PROPOSE, h, e_i, v'⟩ ∧ valid(v') ∧ sendByProposer(h, e_i, v') then
23:       validValue_i ← v'
24:     if ∃v_d, e_d : 2f + 1⟨VOTE, h, e_d, v_d⟩ ∧ valid(v_d) ∧ decision_i = nil then
25:       decision_i ← v_d
26:     else
27:       e_i ← e_i + 1
28:       v_i ← nil
29:       if validValue_i ≠ nil then
30:         proposal_i ← validValue_i
31:       else
32:         proposal_i ← getValue()
```

to that value then it checks if a value $v'$ pre-proposed by the proposer of the current epoch is valid and has at least $2f + 1$ votes, if it is the case, then $p_i$ decides $v'$ and goes to the next height; otherwise it increases the epoch number and updates the value of $proposal_i$ with respect to $validValue_i$.

### 3.2 Byzantine Eventual Synchronous System

This section presents the Algorithm 1 and Algorithms 4 - 5 that solve Consensus in an eventually synchronous model in presence of Byzantine faulty validators. This algorithm has been reported in an early version of [12] with the bugs fixed in [30]. To achieve the consensus in this setting two additional variables need to be used, (i) $lockedEpoch_i$ is an integer representing the last epoch where validator $p_i$ updated $lockedValue_i$, and (ii) $validEpoch_i$ is an integer which represents the last epoch where $p_i$ updates $validValue_i$. These two new variables are used to not violate the agreement property during the asynchronous period. During such period different epochs may overlap at different validators, then it is needed to keep track of the relative epoch when a validator locks in order to not accept "outdated" information generated during a previous epoch. Moreover, a round

**Algorithm 4** Tendermint Consensus part 1 for height $h$ executed by $p_i$

```
1:  Initialization:
2:    e_i := 0                                                    /* Current epoch number */
3:    decision_i := nil                  /* This variable stocks the decision of the validator p_i */
4:    lockedValue_i := nil; validValue_i := nil
5:    lockedEpoch_i := −1; validEpoch_i := −1
6:    proposal_i := getValue() /* This variable stocks the value the validator will (pre-)propose */
7:    v_i := nil                          /* Local variable stocking the pre-preposal if delivered */
8:    validEpoch_j := nil                   /* Local variable stocking the proposer's validEpoch */
9:    vote_i := nil                  /* This variable stock the value the validator will vote for */
10:   timeoutPrePropose := Δ_Pre-propose; timeoutPropose := Δ_Propose; timeoutVote := Δ_Vote

11: Round PRE-PROPOSE:
12:   Send phase:
13:     if decision_i ≠ nil then
14:       ∀v, p_j : (⟨VOTE, h, e_i, v⟩, p_j) ∈ messagesSet_i, broadcast⟨VOTE, h, e_i, v⟩
15:       return
16:     if proposer(h, e_i) = p_i then
17:       broadcast ⟨PRE − PROPOSE, h, e_i, proposal_i, validEpoch_i⟩
18:   Delivery phase:
19:     set timerPrePropose to timeoutPrePropose
20:     while (timerPrePropose not expired) ∧ ¬(∃v_j, e_j : sendByProposer(h, e_i, v_j, e_j)) do
21:       if ∃v_j, e_j : sendByProposer(h, e_i, v_j, e_j) then
22:         v_i ← v_j                                          /* v_j is the value sent by the proposer */
23:         validEpoch_j ← e_j                          /* e_j is the validEpoch sent by the proposer */
24:     if ¬(∃v, epochProp : sendByProposer(h, e_i, v, epochProp)) then
25:       timeoutPrePropose ← timeoutPrePropose + 1
26:   Compute phase:
27:     if 2f + 1 ⟨PROPOSE, h, validEpoch_j, v_i⟩ ∧ validEpoch_j ≥ lockedEpoch_i ∧ validEpoch_j <
        e_i ∧ valid(v_i) then
28:       proposal_i ← v_i
29:     else
30:       if !valid(v_i) ∨ (lockedEpoch_i > validEpoch_j ∧ lockedValue_i ≠ v_i) then
31:         proposal_i ← nil                          /* Note that valid(nil) is set to false */
32:       if valid(v_i) ∧ (lockedEpoch_i = −1 ∨ lockedValue_i = v_i) then
33:         proposal_i ← v_i
```

duration management mechanism needs to be introduced, i.e. increasing timeouts. In the previous algorithm, rounds were lasting $\delta$, the known message delay. In an eventually synchronous system such approach is not feasible, since during the asynchronous period messages may take unbounded delay before being delivered. It follows that, since there are at most $f$ Byzantine faulty validators, when a validator delivers messages from $n - f$ different validators it can terminate the delivery phase, but such phase may last an unbounded time. On the contrary, in the PRE-PROPOSE round only the proposer is sending a message, and generally messages may take a lot of time before being delivered, for such reasons timeouts need to be used in order to manage the rounds duration and adapted to message delays, such that once the system enters in the synchronous period, rounds last enough for messages send during the round to be delivered before the end of it.

The algorithm proceeds in 3 rounds for any given epoch $e$ at height $h$. The description is mainly the same as in Section 3.1, thus in the following we underline just the differences:

**-** Round PRE-PROPOSE (lines 11 - 33, Algorithm 4): The description of this round is mainly the same as before. We highlight the fact that a correct validator

**Algorithm 5** Tendermint Consensus part 2 for height $h$ executed by $p_i$

```
1: Round PROPOSE:
2:   Send phase:
3:     if proposal_i ≠ nil then
4:       broadcast ⟨PROPOSE, h, e_i, proposal_i⟩
5:     broadcast ⟨HeartBeat, PROPOSE, h, e_i⟩
6:   Delivery phase:
7:     set timerPropose to timeoutPropose
8:     while (timerPropose not expires) ∧ ¬(2f + 1⟨HeartBeat, PROPOSE, h, e_i⟩) do{}   /* Note
       that the HeartBeat messages should be from different validators */
9:     if ¬(2f + 1⟨HeartBeat, PROPOSE, h, e_i⟩) then
10:      timeoutPropose ← timeoutPropose + 1
11:  Compute phase:
12:    if ∃v' : 2f + 1⟨PROPOSE, h, e_i, v'⟩ ∧ valid(v') ∧ sendByProposer(h, e_i, v') then
13:      lockedValue_i ← v'
14:      lockedEpoch_i ← e_i
15:      validValue_i ← v'
16:      validEpoch_i ← e_i
17:      vote_i ← v'
18:    else
19:      vote_i ← nil

20: Round VOTE:
21:   Send phase:
22:     ∀v, p_j : (⟨PROPOSE, h, e_i, v⟩, p_j) ∈ messagesSet_i, broadcast⟨PROPOSE, h, e_i, v⟩
23:     if vote_i ≠ nil then
24:       broadcast ⟨VOTE, h, e_i, vote_i⟩
25:     broadcast ⟨HeartBeat, VOTE, h, e_i⟩
26:   Delivery phase:
27:     set timerVote to timeoutVote
28:     while (timerVote not expires) ∧ ¬(2f + 1⟨HeartBeat, VOTE, h, e_i⟩) do{}
29:     if ¬(2f + 1⟨HeartBeat, VOTE, h, e_i⟩) then
30:       timeoutVote ← timeoutVote + 1
31:   Compute phase:
32:     if ∃v'' : 2f + 1⟨PROPOSE, h, e_i, v''⟩ ∧ valid(v'') ∧ sendByProposer(h, e_i, v'') then
33:       validValue_i ← v''
34:       validEpoch_i ← e_i
35:     if ∃v_d, e_d : 2f + 1⟨VOTE, h, e_d, v_d⟩ ∧ valid(v_d) ∧ decision_i = nil then
36:       decision_i ← v_d
37:     else
38:       e_i ← e_i + 1
39:       v_i ← nil
40:       if validValue_i ≠ nil then
41:         proposal_i ← validValue_i
42:       else
43:         proposal_i ← getValue()
```

$p_i$ takes into account also $lockedEpoch_i$ in order to accept a pre-proposed value.
- Round PROPOSE (lines 1 - 19, Algorithm 5): When a correct validator $p_i$ updates $lockedValue_i$ (resp. $validValue_i$), it also update $lockedEpoch_i$ (resp. $validEpoch_i$) to the current epoch.
- Round VOTE (lines 20 - 43, Algorithm 5): If a correct validator $p_i$ delivered at least $f + 1$ same type of messages from an epoch higher than the current one, $p_i$ moves directly to the PRE-PROPOSE round of that epoch and when a correct validator $p_i$ updates $validValue_i$, it also update $validEpoch_i$ to the current epoch.

We recall that each validator has a time-out for each round. If during a round validator $p_i$ does not deliver at least $2f + 1$ messages sent during that round (or the pre-proposal for the PRE-PROPOSE round), the corresponding time-out is increased. Those messages can be values or heartbeats, in the case in which a correct validator has not a value to propose or vote.

### 3.3 Correctness Proof of Tendermint Algorithm in a Byzantine Eventual Synchronous Setting

In this section, we prove the correctness of Algorithm 4 - 5 (Tendermint) in an eventual synchronous system. Due to the lack of space, the missing proofs can be found in the technical report [7].

**Lemma 1 (Validity).** *In an eventual synchronous system, Tendermint verifies the following property: A decided value satisfies the predefined predicate denoted as* valid().

**Lemma 2 (Integrity).** *In an eventual synchronous system, Tendermint verifies the following property: No correct validator decides twice.*

**Lemma 3.** *Let $v$ be a value, $e$ an epoch, and the set $L^{v,e} = \{p_j : p_j$ correct $\wedge$ $lockedValue_j = v \wedge lockedEpoch_j = e$ at the end of epoch $e\}$. In an eventual synchronous system, Tendermint verifies the following property: If $|L^{v,e}| \geq f+1$ then no correct validator $p_i$ will have $lockedValue_i \neq v \wedge lockedEpoch_i \geq e$, at the end of each epoch $e' > e$, moreover a validator in $L^{v,e}$ only proposes $v$ or nil for each epoch $e' > e$.*

**Lemma 4 (Agreement).** *In an eventual synchronous system, Tendermint verifies the following property: If there is a correct validator that decides a value $v$, then eventually all the correct validators decide $v$.*

**Lemma 5 (Termination).** *In an eventual synchronous system, Tendermint verifies the following property: Every correct validator eventually decides some value.*

**Proof** By construction, if a correct validator does not deliver more than $2f + 1$ messages (or 1 from the proposer in the PRE-PROPOSE round) from different validators during the corresponding round, it increases the duration of its round,

so eventually during the synchronous period of the system all the correct validators will deliver the pre-proposal, proposals and votes from correct validators respectively during the PRE-PROPOSE, PROPOSE and the VOTE round. Let $e$ be the first epoch after that time.

If a correct validator decides before $e$, by Lemma 4 all correct validators decide which ends the proof. Otherwise at the beginning of epoch $e$, no correct validator decides yet. Let $p_i$ be the proposer of $e$. We assume that $p_i$ is correct and pre-propose $v$; $v$ is valid since $getValue()$ always return a valid value (lines 6, Algorithm 4 & line 43, Algorithm 5), and $validValue_i$ is always valid (lines 12 & 32, Algorithm 5). We have 2 cases:

- Case 1: At the beginning of epoch $e$, $|\{p_j : p_j \text{ correct} \wedge (lockedEpoch_j \leq validEpoch_i \vee lockedValue_j = v)\}| \geq 2f + 1$.
  Let $p_j$ be a correct validator where the condition $lockedEpoch_j \leq validEpoch_i \vee lockedValue_j = v$ holds. After the delivery of the pre-proposal $v$ from $i$, $p_j$ will update $proposal_j$ to $v$ (lines 27 - 33, Algorithm 4). During the PROPOSE round, $p_j$ proposes $v$ (line 4, Algorithm 5), and since there are at least $2f + 1$ similar correct validators they will all propose $v$, and all correct validators will deliver at least $2f + 1$ proposals for $v$ (line 7, Algorithm 5). Correct validators will set their *vote* to $v$ (lines 12 - 4, Algorithm 5), will vote $v$, and will deliver these votes, so at least $2f + 1$ of votes (lines 24 & 26, Algorithm 5). Since we assume that no correct validators decided yet, and since they deliver at least $2f + 1$ votes for $v$, they will decide $v$ (lines 35 - 36, Algorithm 5).
- Case 2: At the beginning of epoch $e$, $|\{p_j : p_j \text{ correct} \wedge (lockedEpoch_j \leq validEpoch_i \vee lockedValue_j = v)\}| < 2f + 1$.
  Let $p_j$ be a correct validator where the condition $lockedEpoch_j > validEpoch_i \wedge lockedValue_j \neq v$ holds. When $p_i$ will make the pre-proposal, $p_j$ will set $proposal_j$ to *nil* (line 31, Algorithm 4) and will propose *nil* (line 4, Algorithm 5).
  By counting only the propose value of the correct validators, no value will have at least $2f + 1$ proposals for $v$. There are two cases:
  - No correct validator delivers at least $2f + 1$ proposals for $v$ during the PROPOSE round, so they will all set their *vote* to *nil*, vote *nil* and go to the next epoch without changing their state (lines 19 & 24 - 26 & 37 - 43, Algorithm 5).
  - If there are some correct validators that delivers at least $2f + 1$ proposals for $v$ during the PROPOSE round, which means that some Byzantine validators send proposals for $v$ to those validators.
    As in the previous case, they will vote for $v$, and since there are $2f + 1$ of them, all correct validators will decide $v$. Otherwise, there are less than $2f + 1$ correct validators that deliver at least $2f + 1$ proposals for $v$. Only them will vote for $v$ (line 24, Algorithm 5). Without Byzantine validators, there will be less than $2f + 1$ vote for $v$, no correct validator will decide (lines 35 - 36, Algorithm 5) and they will go to the next epoch, if Byzantine validators send votes for $v$ to a correct validator

such as it delivers at least $2f + 1$ votes for $v$ during VOTE round, then it will decide (lines 35 - 36, Algorithm 5), and by Lemma 4 all correct validators will eventually decide.
Let $p_k$ be one of the correct validators that delivers at least $2f + 1$ proposals for $v$ during PROPOSE round, it means that $lockedValue_k = v$ and $lockedEpoch_k = e$. It follows that at the end of epoch $e$, all correct validators will have $validValue = v$ and $validEpoch = e$.

If there is no decision, either no correct validator changes its state, otherwise all correct validators change their state and have the same $validValue$ and $validEpoch$, eventually a proposer of an epoch will satisfy the case 1, and that ends the proof.

If $p_i$, the proposer of epoch $e$, is Byzantine and more than $2f+1$ correct validators delivered the same message during PRE-PROPOSE round, and the pre-proposal is valid, the situation is like $p_i$ was correct. Otherwise, there are not enough correct validators that delivered the pre-proposal, or if the pre-proposal is not valid, then there will be less than $2f+1$ correct validators that will propose that value, which is similar to the case 2.

Since the proposer is selected in a round robin fashion, a correct validator will eventually be the proposer, and correct validators will decide. $\square_{Lemma\ 5}$

**Theorem 1.** *In an eventual synchronous system, Tendermint implements the consensus specification.*

### 3.4 Complexity of Tendermint Algorithm in a Byzantine Eventual Synchronous Setting

Let us consider the following scenario after the asynchronous period (i.e., after $\tau$), in which in the first $f$ epochs, $e_{i+1}, \ldots, e_{i+f}$, there are $f$ Byzantine proposers that make lock only one correct validator at each epoch on $f$ different values with different $lockedEpoch$, $e_{i+1}, \ldots, e_{i+f}$. Let $p_j$ be the last correct validator that locked, and let $v$ such value ($lockedValue_j = v$) with $lockedEpoch_j = e_{i+f}$. Then all the other correct validators have $validValue$ set to $v$ and $validEpoch$ set to $e_{i+f}$. This happens thanks to the fact that when a correct validator locks on a value then at the end of the epoch every correct validator sets its $validValue$ to that value. The algorithm terminates when a pre-proposal is proposed and voted by more than $2f$ correct validators, i.e, when the pre-proposed value has $validEpoch$ greater equal than the validator $lockedEpoch$. Thus, during the period of synchrony, the first correct proposer that proposes leads the algorithm to terminate in $f + 1$ rounds. Let us consider the case in which there $f$ correct validators locked on $f$ different values with different $lockedEpoch$ before $\tau$. Let us assume that $p_j$ is the last correct validator that locked on a value $v$, thus it has the highest $lockedEpoch$ but not all the correct validators have their $validValue$ set to $v$ (due to the asynchronous communication). Let us now consider that after $\tau$ the first $f$ proposers are Byzantines and stay silent. The following proposers are correct but their pre-propose value might not be accepted

by enough correct validators as long as $p_j$, with the highest *validEpoch* and *lockedEpoch* proposes. Which eventually happens due to the round robin selection function. Thus, the protocol terminates in a number of epochs proportional to the number of validators $O(n)$, while the lower bound to solve BFT Consensus in the worst case scenario is $f + 1$ [20]. As for message complexity, since at each epoch, all validators broadcast messages, it follows that during one epoch the protocol uses $O(n^2)$ messages, thus in the worst case scenario the message complexity is $O(n^3)$.

In the following we address the bit complexity of Tendermint. In Tendermint, each message is composed as follow:

- `PRE-PROPOSE`: The marker that the message is from the round `PRE-PROPOSE`; two integers one for the current height, and the second for the current epoch; the proposed value; and an integer representing the epoch on which the proposer last updated its $validValue$.
- `PROPOSE`: The marker that the message is from the round `PROPOSE`; two integers representing the current height and the current epoch; and a value which is the proposed block.
- `VOTE`: The marker that the message is from the round `VOTE`; two integers representing the current height and the current epoch; and a value which is the voted block.
- HeartBeat: The marker that the HeartBeat is from the round `VOTE` or `PROPOSE`; two integers representing the current height and the current epoch.

A correct validator keeps in memory, for each epoch for a given height, one message for each type (`PROPOSE`, `VOTE`) and at most 2 messages of type HeartBeat from each validator, and only one `PRE-PROPOSE`. A correct validator may have at most 1 message from `PRE-PROPOSE`, $n$ messages from `PROPOSE`, $n$ messages from `VOTE`, and $2n$ messages of type HeartBeat. Hence, for each epoch at any given height, a validator stores at most 4n+1 messages of size $O(\log n)$. In the worst case, for the whole execution, a validator may store $O(n^2)$ messages. Therefore, the bit complexity in the worst case is $O(n^2 \log n)$.

Note that [24] proposes a bit complexity of $O(n^3 \log n)$ for an optimal round complexity using a variant of the tree structure of the Exponential Information Gathering protocol introduced in [22]. Clearly, there is a tradeoff between the bit complexity and the round complexity of the Byzantine agreement.

## 4 Conclusion

The contribution of this work is twofold. First, it analyzes Tendermint consensus protocol and provides detailed proof of its correctness and complexity. Second, it dissects such protocol in order to link the algorithmic techniques to the considered system model. We believe that this methodology can contribute in making Byzantine-tolerant consensus algorithms more understandable for developers and practitioners.

**Acknowledgment**

The authors would like to thank the reviewers of NETYS 2019 for their insightful comments. The authors also thank Zaynah Dargaye for numerous discussions, and in particular for the consistency of this work.

## References

1. Livelock scenario. `https://github.com/tendermint/tendermint/wiki/0.7-Livelock-Scenario`, accessed: 2019-03-14
2. Abraham, I., Gueta, G., Malkhi, D.: Hot-stuff the linear, optimal-resilience, one-message BFT devil. CoRR **abs/1803.05069** (2018), `http://arxiv.org/abs/1803.05069`
3. Abraham, I., Malkhi, D., Nayak, K., Ren, L., Spiegelman, A.: Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. CoRR, abs/1612.02916 (2016)
4. Aguilera, M.K.: A pleasant stroll through the land of infinitely many creatures. ACM Sigact News **35**(2), 36–59 (2004)
5. Amoussou-Guenou, Y., Del Pozzo, A., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Correctness and Fairness of Tendermint-core Blockchains. CoRR **abs/1805.08429** (2018)
6. Amoussou-Guenou, Y., Del Pozzo, A., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Correctness of Tendermint-Core Blockchains. In: 22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China. pp. 16:1–16:16 (2018)
7. Amoussou-Guenou, Y., Del Pozzo, A., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Dissecting Tendermint. Research report, LIP6, Sorbonne Université, CNRS, UMR 7606 ; CEA List (2018), `https://hal.archives-ouvertes.fr/hal-01881212v2`
8. Anceaume, E., Del Pozzo, A., Ludinard, R., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Blockchain Abstract Data Type. To appear, SPAA 2019 (2019)
9. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A.D., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S.W., Yellick., J.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018. pp. 30:1–30:15 (2018)
10. Baldoni, R., Bertier, M., Raynal, M., Tucci-Piergiovanni, S.: Looking for a definition of dynamic distributed systems. In: International Conference on Parallel Computing Technologies. pp. 1–14. Springer (2007)
11. Ben-Or, M.: Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: Proceedings of the second annual ACM symposium on Principles of distributed computing. pp. 27–30. ACM (1983)
12. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on bft consensus. Tech. rep., Tendermint (2018), `https://arxiv.org/abs/1807.04938`
13. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols (extended abstract. In: in Advances in Cryptology: CRYPTO 2001. pp. 524–541. Springer (2001)
14. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI) (1999)

15. Crain, T., Gramoli, V., Larrea, M., Raynal, M.: (Leader/Randomization/Signature)-free Byzantine Consensus for Consortium Blockchains. http://csrg.redbellyblockchain.io/doc/ConsensusRedBellyBlockchain.pdf (visited on 2018-05-22) (2017)
16. Crain, T., Gramoli, V., Larrea, M., Raynal, M.: Dbft: Efficient byzantine consensus with a weak coordinator and its application to consortium blockchains. arXiv preprint arXiv:1702.03068 (2017)
17. Decker, C., Seidel, J., Wattenhofer, R.: Bitcoin Meets Strong Consistency. In: Proceedings of the 17th International Conference on Distributed Computing and Networking Conference (ICDCN) (2016)
18. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
19. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings. pp. 139–147 (1992)
20. Fischer, M.J., Lynch, N.A.: A lower bound for the time to assure interactive consistency. Information processing letters **14**(4), 183–186 (1982)
21. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
22. Garay, J.A., Moses, Y.: Fully polynomial byzantine agreement in t+1 rounds. In: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA. pp. 31–41 (1993)
23. Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In: Proceedings of the 25th USENIX Security Symposium (2016)
24. Kowalski, D.R., Mostéfaoui, A.: Synchronous byzantine agreement with nearly a cubic number of communication bits: synchronous byzantine agreement with nearly a cubic number of communication bits. In: ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013. pp. 84–91 (2013)
25. Kwon, J., Buchman, E.: Cosmos: A Network of Distributed Ledgers. https://cosmos.network/resources/whitepaper (visited on 2018-05-22)
26. Kwon, J., Buchman, E.: Tendermint. https://tendermint.readthedocs.io/en/master/specification.html (visited on 2018-05-22)
27. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3), 382–401 (Jul 1982)
28. Muñoz-Escoí, F.D., de Juan-Marín, R.: On synchrony in dynamic distributed systems. Open Computer Science **8**(1), 154–164 (2018). https://doi.org/10.1515/comp-2018-0014, https://doi.org/10.1515/comp-2018-0014
29. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM **27**(2), 228–234 (Apr 1980)
30. Tendermint: Tendermint: correctness issues. https://github.com/tendermint/spec/issues (visited on 2018-09-24)
31. Tendermint: Tendermint: Tendermint Core (BFT Consensus) in Go. https://github.com/tendermint/tendermint/blob/e88f74bb9bb9edb9c311f256037fcca217b45ab6/consensus/state.go (visited on 2018-05-22)